Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 53 (2004) 259–284
Science of Computer Programming
www.elsevier.com/locate/scico

A customizable approach to full lifecycle variability management

Klaus Schmid*, Isabel John

Fraunhofer Institute for Experimental Software Engineering (IESE), Sauerwiesen 6, D-67661 Kaiserslautern, Germany

Received 12 January 2003; received in revised form 10 April 2003; accepted 24 April 2003

Available online 6 August 2004

Abstract

In order to enable a smooth transition to product line development for an organization that so far only performed single system development, it is necessary to keep as many of the existing notations and approaches in place as possible.

This requires adaptability of the basic variability management approach to the specific situation at hand. In this paper we describe an approach that enables homogenous variability management across the different lifecycle stages, independent of the specific notation. The approach is accompanied by a meta-model and a process for introducing the variability management approach by developing a notation-independent representation. This approach has so far been applied in several cases where our Product Line engineering method PuLSE™ has been introduced into a software development organization.

1. Introduction

Variability Management is a concern that arises in Product Line Engineering throughout all lifecycle phases. It can actually be seen as *the* key feature that distinguishes Product Line Engineering from other approaches to software development [9].

Product Line Engineering [4,10,49] is an approach that aims at exploiting reuse potential between products developed in an organization by identifying the commonalities between the products and systematizing the variabilities. In Product Line Engineering,

* Corresponding author. Tel.: +49-6301-707-158/-250.
E-mail addresses: Klaus.Schmid@iese.fraunhofer.de (K. Schmid), Isabel.John@iese.fraunhofer.de (I. John).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2003.04.002

variabilities have to be identified, modeled, stored, resolved, instantiated and changed. So, variabilities have to be managed throughout the whole product line engineering lifecycle.

This requires a comprehensive approach to the management of variability that can be applied throughout the various lifecycle stages, their artifacts, and their accompanying notations in an universal manner. Moreover, in order to enable a smooth transition to product line development for an organization that so far only performed single system development, it is necessary to keep as many of the existing notations and approaches in place as possible. For this reason, we developed a customizable approach to full lifecycle variability management. This allows to practically apply the approach in a wide range of industrial settings. This is particularly motivated by our industrial projects using the PuLSE-approach [3,4], as there we needed an approach that enables us to homogeneously manage variability across the different lifecycle stages, independent of the specific notation. PuLSE is a modular and customizable product line engineering method that provides a complete framework that covers the whole software product line development lifecycle, including reuse infrastructure construction, usage, and evolution.

In this paper, we will discuss this approach to variability management including the meta-model it is based on and the fundamental assumptions and concepts on which it relies. We will describe a process for introducing our variability management approach by developing a representation-specific representation and describe our experiences with the approach in different areas like textual modeling, the Unified Modeling Language UML™ or implementation.

The paper is structured as follows. In Section 2 we describe the requirements on a variability management approach, including the elements such an approach needs to have and the requirements such an approach has to support. In Section 3 we describe related work on variability management approaches supporting variability in different stages of the software development lifecycle. In Section 4 we present our approach to systematic variability management and provide the basis for this approach with the meta-model described in Section 5. We present the process for developing a representation-specific variability description approach in Section 6, share our experiences with different applications of the variability management approach and the process in Section 7 and conclude the paper in Section 8.

2. Requirements on the approach

In order to define a systematic approach for variability management, we need to first define what our interpretation of variability management is. Throughout the paper, we will use the following definition:

> *Variability management* encompasses the activities of explicitly representing variability in software artifacts throughout the lifecycle, managing dependences among different variabilities, and supporting the instantiation of the variabilities.

Using this definition we can regard variability management as THE necessary addition that arises in the representation of product lines over single system concerns.

In our practical work, we are focusing on the introduction of a product line approach in an organization that so far performed only single system development. This is a quite common situation in the context of Fraunhofer IESE where we do technology transfer to different contexts. In order to facilitate the introduction of the variability concepts we like to keep the existing notations and processes as far as possible in place. So, for example, if an organization performs a text-based requirements process, we may often keep the text-based process and augment it with variability concepts. Later on, after the variability issues have been widely understood and accepted we then move to a more formal notation like the UML. In other cases we developed an extension of a graphical notation, to which we added variability concepts by adding notational elements along with a decision model [40].

On the other hand we must support variability throughout the lifecycle. Thus, it is necessary to map the same variability in a consistent manner to the various artifacts like requirements or code, in order to widely apply it in industrial practice.

For example, when we introduced Product Line engineering methods in the company context described in Section 7.1, we had to consider the following factors. In order to describe variability between the planned products we had to integrate variability into the requirements, the architecture and the implementation. We had to make sure that the developers and engineers understand and use the notation and had to provide mechanisms for deriving applications. When for example modeling requirements on the printing domain of the systems of this company we had to consider different binding times for different requirements (e.g. the printer drivers have to be determined at compile time, the decision for a certain page width can be left open until installation time or even until run time). We also had to consider and model different relations between the possible variation points (e.g. if a certain printer driver is selected the page width cannot be larger than 58 mm).

This breadth of issues that must be addressed together with the fact that we need to be able to address basically any kind of underlying process and notations led us to develop a very general approach to variability management.

In particular, we see five main issues that should be addressed by a variability management technique:

1. The definition of variation points in the base artifacts.
2. The definition of elements that can be potentially bound to these variation points.
3. The definition of the relation among variation points and potentially bound elements.
4. The definition of constraints among potential variation point bindings, that restrict the potential instantiations.
5. A selection mechanism to define the elements that should go into a specific product. This selection mechanism can support manual selection but also be tool supported.

Any comprehensive approach to variability management must support these requirements. However, many approaches are very restricted in their support of some of these requirements.

Besides these basic technical requirements addressing the notation, we derived some further requirements on a variability management approach from our experience in industrial transfer projects:

- *The approach must be notation-independent.*
  This requirement aims at supporting variability management in a variety of contexts.

- *The approach must be applicable to all kinds of lifecycle artifacts.*
  In order to perform a homogeneous and particularly traceable management of artifacts throughout the lifecycle we aim at an approach that can be systematically applied throughout the lifecycle.
- *The approach must support traceability of variabilities both horizontally and vertically:*
  - horizontally means that we must be able to trace a variability to the various places within a lifecycle artifact, where it has an impact;
  - vertically means that we want to be able to trace a variability in one lifecycle stage to corresponding variabilities in artifacts of other lifecycle stages.

  In order to systematically maintain and evolve variability, we must be able to trace any impact of variability on the artifacts we are building.
- *It must be possible to hierarchically structure the approach*.
  This is necessary in order to keep the approach scalable.

We developed an approach that satisfies these requirements. This approach draws on earlier work like [15,39,40] and is based in a meta-model for variability management that describes how the artifacts that we use are built and instantiated.

3. Related work

Systematic variability management has been an issue within product line engineering approaches for some time. Different approaches exist that differ in their degree of coverage of lifecycle stages (for one stage, multi-stage or full lifecycle support) and in their notation dependency (general, notation-independent approaches or specific approaches that use a fixed notation).

Different proposals have been made for full lifecycle management of variability using decision models [2,27]. However, these approaches are always related to a specific notation. There are three exceptions we found so far: the Synthesis approach [28,41], which is described in a manner independent of the specific notation, the approach presented by Muthig in [35], which provides a notation-independent meta-model, and the SPLIT-method [14] that can be used with different notations.

One of the key principles of the Synthesis approach is abstraction-based reuse [28], which provides a means for representing a product family as a set of adaptable work products and for deriving instances of those work products that represent a particular system. An adaptable work product concisely represents a set of similar items that vary in well defined ways. Within Synthesis, a decision modeling process is integrated, that defines the abstract form (concepts, decisions, and structure) of an application modeling notation. A decision model defines the set of requirements and engineering decisions that an application engineer must resolve in order to describe and construct a product and determine the extent of variation that is possible among the systems of the domain. Equivalently, these decisions determine the form and content of the work products that comprise each system. With the help of the decision model, a set of presentation paradigms is identified by considering the ways domain experts generally represent various problem facets in a manner consistent with the decision hierarchy defined in the decision model. Different from the approach described here, the decision model has to exist in order to

build a representation of variability, there are no binding times specified in the model and the decision model in Synthesis does not provide means for a hierarchical description of variabilities and decisions. Furthermore, an explicit tool support is not given in the synthesis approach.

The approach of Muthig [35] that has influenced our approach describes an approach for notation-independent introduction of product lines. The approach is a general and lightweight product line approach, including the description of general product line artifacts, including a decision model. The decision model is built up and used during the domain engineering phase and instantiated during the application engineering phase. Different to the approach we describe here, the approach by Muthig focuses more on selecting an existing approach that is suitable for the context of an organization that wants to introduce product line engineering than on extending existing modeling and implementation mechanisms with elements that make it possible to model and use variability. The approach by Muthig has been validated in different projects and in a case study and is also strongly related to the PuLSE method. The approach by Muthig and the approach we describe here complement each other in a way that the approach by Muthig focuses more on the incremental introduction, so on the phase of transition to product lines, whereas our approach focuses more on using product line methods with a comprehensive variability modeling approach.

In the SPLIT approach [13,14], no particular notation for the description of product line assets is assumed; the approach was used with an extension of UML as object technology that has proven to be useful and efficient in the context of SPLIT. SPLIT uses a multi-level decision model that defines rules for centralizing decisions needed to unequivocally identify one single product of the product line by making relations between decisions explicit. The global decision model is a hierarchical decomposition of the variation point level, the asset level and the core asset level. Different to our approach, no support is given to apply the SPLIT approach in an environment that does not use UML. Although SPLIT is intentionally described as notation independent, it strongly relates to UML.

An approach that covers more lifecycle stages and is partially independent of a fixed notation but uses XML as the underlying representation is the approach by Becker [5]. The approach uses a general model to express variability in product lines and instantiates the model with the help of VSL and XML. With the help of this model and the transformations described in his work, it is possible to express variability in the whole lifecycle. Anyway, the artifacts built with the approach have to support XML extensions and are not possible to use in an environment without XML support, e.g. where no tool support is needed or possible. So, although this approach supports full lifecycle variability management it is not totally independent of the notation. This approach has been applied in the area of embedded systems for automobiles and was used in a tooling environment with Rational Reqisite Pro and Rational Rose extended with their own tools [48].

Several approaches exist that do variability modeling and management for a single lifecycle stage or for two stages and that use a specific notation. Approaches exist that describe how to extend existing modeling, design and implementation mechanisms in order to express variability. In most cases those mechanisms are strongly related to the specific technique that is extended. The techniques can be distinguished into text-based

extensions, extensions that are based on different parts of UML, dealing with requirements, architecture or design models and techniques extending implementation mechanisms.

Text-based techniques either suggest a special structure for writing down the requirements on a product line, like the commonality analysis [49] or they extend textual use cases [7,25]. UML-based techniques [2,7,20,25,31,32,34,45] are proposed. Feature-oriented approaches like FODA [27] or specific design-based approaches have been proposed [8,17], and of course, implementation mechanisms have been studied [1,18,23, 36]. The approach by Svahnberg and van Gurp [43,46] covers more than one lifecycle but takes only architectural and implementation variability into account and describes an approach that uses features as the basic entity of variability, so the approach is notation independent on the architecture and implementation side but by using features it is not independent of the requirements notation. Furthermore, binding times earlier than architectural binding are ignored.

Variability Management has also been studied in areas other than software engineering. For example, configuration-based approaches like [29] exist. The dynamic constraint solving problem DCSP [33] is strongly related to variability management. Techniques in this area have been applied for the configuration of technical systems [19,24]. The applicability of constraint techniques for variability modeling has been recognized [21] and is currently being investigated, e.g. in the ConIPF project [12]. Different techniques for handling the different problem areas in variability management have been identified [21]. For example, phases and strategies can be used to represent binding times in order to support product derivation and compositional relations can be used to represent relations between variation points. Although the applicability of different techniques has been identified there is no integrated approach for variability management yet, only first ideas have been presented [22]. Constraint techniques can provide support in the systematic derivation of products but do not give support on the whole process of variability management, like, for example, finding a mapping from an existing representation to the variability notation or capturing different binding times.

The approach that we present here overcomes the shortcomings of existing approaches by proposing a notation-independent approach to full lifecycle variability management.

4. Systematic variability management

An approach to comprehensive variability management needs to address from our point of view the different requirements defined in Section 2.

Of course there are many different ways of supporting variability management on the different levels of artifacts. A taxonomy and a good overview of such approaches can be found in [43].

The approach we will discuss in this section is characterized by the following aspects:

- Decisions on variability can be expressed on a high level, i.e., independent of the specific variation in an artifact.
- Decisions can be described in a way that is on a rather abstract level, enabling people to describe variability in a very general manner.

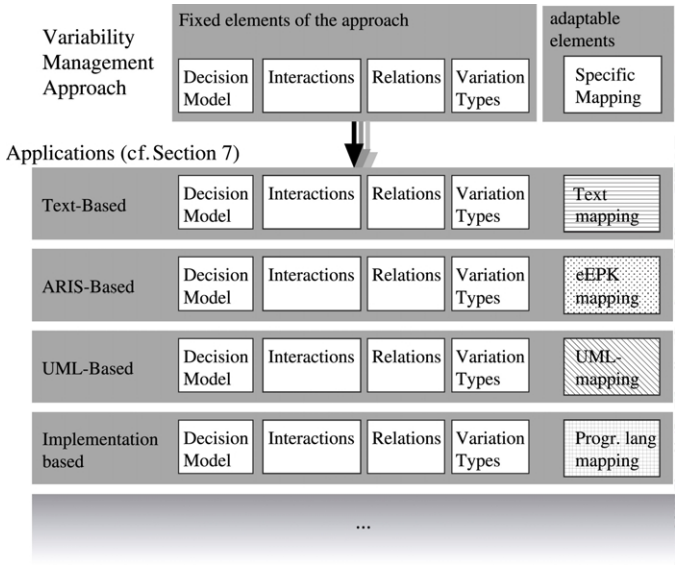

Fig. 1. The variability management approach.

The combination of these aspects differentiates this approach from most other approaches to variability management.

The specific approach to variability management we propose consists of the following components (cf. Fig. 1):

1. A decision model as a basis for characterizing the effects of variability.
2. A mechanism to describe interactions among different decisions.
3. A mechanism to describe the relation between variation points and the specific decisions (or group of decisions) on which the resolution of the variability depends.
4. A common (maximal) set of selector types.
5. An accompanying mapping of the selector types to the specific notation to express the variation points in the artifacts.

Only the last point, the mapping, has to be adapted to the specific representation technique. The other parts as well as the semantic interrelation among them are independent of the specific representation approach. This situation is illustrated in Fig. 1. The first four parts of the approach are identical in all applications, only the specific mapping must be adapted and modified. For example, in the example of modeling a printer domain in an embedded systems context mentioned in Section 2, we can use the notation that is used in the organisation (textual requirements in a use case related form) and just find a specific mapping to the textual representation. The other elements of the approach are fixed. So, the developers and engineers can model and use the modeling artefacts almost as before, only the variation points have to be specified and entered into the decision model.

By centralizing all variability-related information in the decision model changes can be localized. Deriving new products, can be controlled with the help of the decision model,

by instantiating all decision variables. So, having an extra model where variability is centralized encapsulates the variability concern throughout the lifecycle.

We describe a process for deriving an adequate mapping in Section 6. Here, we will describe how the five components of the approach are implemented by our approach.

4.1. The decision model

The decision model was initially devised in the context of the Synthesis approach for variability management [28]. In the meantime, this technique has been widely applied both in research and in industry [2,27,29,31,34,35,40].

The specific kind of decision model we propose is different from other approaches in two ways:

- It is more comprehensive in terms of the information it contains.
- It does not explicitly relate to the variation points, but rather it defines decision variables which are then referenced at the specific variation points using the decision evaluation primitives.

The second characteristic makes this approach particularly notation independent.

A decision variable is a unique identifier for a variation. A decision variable can be used for several variation points (e.g. the decision variable PRINTER_DRIVER is used at each variation point where the printer driver has to be chosen). Each of the decision variables that is defined in the decision model is in turn described by the following information:

- **Name:** The *name* of the defined decision variable; the name must be unique in the decision model.
- **Relevancy:** The *relevancy* of a decision variable for an instantiation may depend on other decision variables, e.g. the decision variable describing the memory size is only valid if the decision variable describing the existence of memory is true. This can be made explicit by the relevancy information.
- **Description:** A textual *description* of the decision captured by the decision variable.
- **Range:** The *range* of values that the decision variable can take on. This can be basically any of the typical data types used in programming languages. However, instead of a real or integer often only a range is important. Moreover, probably the most common type is the enumeration, as the relevant values are often domain dependent. Further, boolean variables are quite common.
- **Cardinality:** As opposed to other approaches, we do not emphasize the difference between variables which can only assume a single value and variables that can assume sets of values during application engineering. Rather, we define a selection criterion, defining how many of the values of a decision variable can be assumed by it. This is due to the fact that in practice we found cases where sets are required, but their cardinality is restricted.

  This is represented by *m–n*, similar to UML associations [37]. Thus, basically, all decision variables get a set of values during application engineering. However, we use *1* as a shorthand notation for *1–1* and in this case we also write the value of the decision variable as a single value (without curly brackets) and treat it for the purpose of decision evaluation like a non-set value.

- **Constraints:** Constraints are used to describe interrelations among different decision variables. This is used to describe value restrictions imposed by the value of one variable onto another variable. We use this approach also to describe the *requires relationship*, as this can be treated as a special case in our framework. The constraints can of course also contain domain knowledge. Consider for example the following constraint: the value of the decision variable describing the memory size has to be >16384 if the decision variable describing the existence of memory is true. This constraint at the same time represents the domain knowledge that in the product line the minimum memory size is 16KB.[1]
- **Binding times:** A list of possible binding times, describing when the decision can be bound. This can be *source time*, *compile time*, *installation time*, etc. Additional binding times may exist, and can be product line specific. As opposed to the FODA work [27] and many related approaches, we allow several binding times, meaning depending on the specific product the variability may be bound at any of these times. This technique was first introduced in ODM [42] as "binding sites". In particular, this implies that a development decision for one system may be an initialization decision for another—a case we found quite frequently in practice.

Depending on the specific context of our industrial projects, we sometimes used slight variations of this approach to decision modeling. However, regarding the information content, it was always a subset of this information. In cases where we want to use the decision model also as a basis for tracking implementation and planned evolution (as far as the evolutionary aspects are described in the decision model), it is useful to define an additional facet, describing the binding times already supported by the implementation. This may of course include "not yet supported" and in general the supported binding times should be later than or equal to the current binding times.

Using this description of a decision variable, we can define a decision model simply as a set of decision variable definitions. For practical reasons this will usually be represented by a table. However, especially for particularly large decision tables this may be impractical. To handle this case hierarchical decision models have been proposed [14]. However, it should be noted that there should be for practicality and consistency reasons only one decision model for all process phases.

There is another reason that may lead to splitting the decision model, which is that certain decisions may be relevant only for certain binding times. In this case it is useful to define subsets of the decision model based on the binding times. Having split the decision model then helps developers in using just the relevant part of the decision model at a certain binding time. In this case it might be useful for some binding times like e.g. for compile time to decompose the model into parts that can be directly implemented, e.g., using makefiles, or preprocessor directives. However, as these notations do not support the full range of information that we require, it will always be necessary to keep additional information in the form of comments in these representations.

The decision model provides the basis for describing the individual products, as a specific product can be defined by assigning values to the decisions in the decision model,

[1] Of course, this would usually be represented with a constant like Min_Mem_Size := 16384.

where the constraints among the decisions determine the possible values. With these assigned decisions the variation points related to the decisions can be instantiated.

This instantiation of variation points can be done manually by going through the list of decision variables and giving values to the variables while respecting the constraints. In order to give tool support here constraint techniques [33] could be helpful in supporting the derivation.

4.2. Decision evaluation primitives

As a basis for describing the relationship, both among different decisions (in the decision model *relevancy* and *constraints*) as well as the relationship of a variation point instantiation to a specific decision, we need to be able to describe complex evaluations of the decisions. The basic constructs for describing these evaluations are called the decision evaluation primitives. These primitives support three tasks:

- The description of the *relevancy* dependences
- The description of the *constraints* dependences
- The description of the relation between variation points instantiations (selector types) and the specific decisions

Thus, they support the components two and three of our approach. It would also be possible to use three different approaches to satisfy these needs, however, it is of course more practical to use an unified approach.

The following list provides some relations we use for decision evaluation:

sub	real subset $\subset$
subeq	subset or equal $\subseteq$
#	cardinality of a set
in	is element of a set
->	logical implication
<->	mutual implication (iff)

In addition logical relations (AND, OR, NOT) are used. Regarding the different contexts in which we use these evaluation primitives we further differentiate:

- For describing the *relevancy* dependency we need to derive a boolean value, i.e., a *logical expression*. If for specific values for a product this value is evaluated as *true*, the corresponding decision variable is relevant for this product. By default all decision variables are relevant. The relevancy relation is only used to eliminate certain decision variables, which are not required in a certain context.
- For describing the *constraints* we need to build *relational expressions*, involving the specific decision variable. Thus, the constraint MEM_PRESENCE=TRUE -> MEM_VALUE > 100 as part of the description of MEM_VALUE would restrict the possible values of MEM_VALUE in case MEM_PRESENCE would be true.
- Finally, for describing the relation between variation points and decision values, we need both *logical expression* and *value expressions* (e.g., integer, enumeration), depending on the specific selector type.

The approach to decision evaluation proposed here is very similar to expression evaluation in existing programming languages or constraint languages, the main extension being that we may need to deal with set values.

4.3. Supported selector types

Many different types of variability have been proposed in literature: optionalities, alternatives, set-optionalities (a set of options may be selected), etc. In our approach we differentiate between the variation point, i.e., the "point" in a generic artifact where variation may occur, and the specific possible instantiations along with the logic for selecting among them. The latter we call a selector.

Based on our practical experience we deem the following types of variability selectors to be the most relevant. They are neither minimal nor do they cover all proposed concepts, but they have been found to be sufficient from a practical point of view:

- **Optionality:** A property either exists in a product or not.
- **Alternative:** Two possible resolutions for the variability exist and for a specific product only one of them can be chosen.
- **Set alternative:** Only a single instance may be selected out of a range of possible alternatives.
- **Set selection:** Several variabilities may be simultaneously selected for inclusion in a product.
- **Value reference:** The value of the decision variable can be directly included in the product line model. (This, of course, only makes sense with decision variables that only assume a single variable in application engineering.)

We chose to support alternative, set alternative and set selection although the first two are a specialisation of set selection due to practicability. Having alternative and set alternative as explicit variability types makes the product line artifacts much shorter and better readable. All these variability selector types are mapped to specific representations in the context of a specific notation. In Section 6 we describe an approach for developing such a mapping. Optionality and alternative use logical expressions to determine the specific instantiation that shall be made. Set alternative, set selection, and value take a value expression as basis. In addition set alternative and set selection take values as labels in order to describe the variabilities that should be part of the instantiation.

A key task of the decision evaluation primitives is to relate a decision to a selector that is used to define the instantiation of a variation point. We usually do not directly relate the impact of a decision variable (like "decision_variable < value" or "decision_variable = TRUE") to the variation points as the same decision may easily have many different forms of impact on the variation points. This allows us to decouple the decision itself from its impact on the product line model (i.e., the specific implementation).

5. The meta-model

The approach we described in the previous section provides a basis for the customization of representation mechanisms. In order to be effectively used, however, it needs

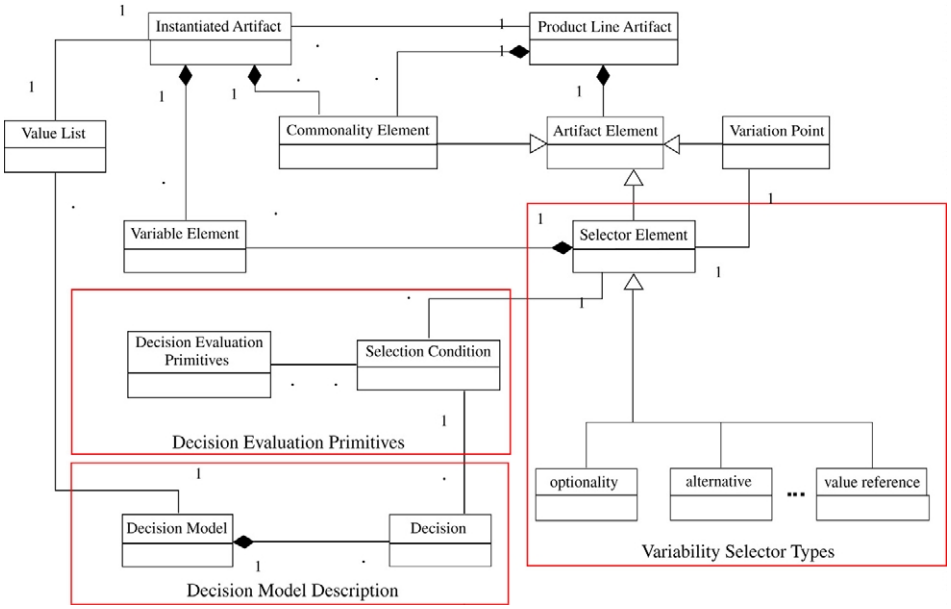

Fig. 2. Representation-independent variability meta-model.

of course to be embedded in an artifact model, that describes how the product line artifacts are built and instantiated. Our model is of course based on existing meta-models like [5,16,35,44]. The model is, however, also unique in several ways:

- It simultaneously describes product line artifacts and their instantiation.
- It explicitly represents the selection process as we described it above.

The meta-model which is given in Fig. 2 describes our representation-independent variability in a UML notation. We can differentiate among elements that are relevant to our specific variability management approach, which are shown in boxes and the remainder of the model, which represents the general artifact relationship.

In the general part of the model there is little unusual to be found, apart from the distinction between instantiated artifact and product line artifact. If we look at the realization of the basic concepts introduced in the previous section, certain interesting observations can be made.

The variability selectors are represented by an explicit selector element. This selector element refers to selection conditions, which are composed in turn of decision evaluation primitives. The selector element also *contains* the variable elements. This implies that the selector elements do not themselves appear in the instantiated artifact, but only the variable elements, which is exactly what is intended. A selector element then refers to a variation point. This is a one-to-one relation, as the contents of a specific selection may in turn depend on the variation point. It could, however, be useful to generalize this relation.

The *decision model* is in turn represented as a collection of decisions, which are referred to by the selection conditions. For determining instantiations we need in addition a value list for these decisions as is shown in the meta-model.

The *decision evaluation primitives* are used to form the selection conditions. Note that we defined the decision evaluation primitives as generic constructs without a relation to specific conditions.

6. Developing a representation-specific variability description approach

The concepts we discussed so far are representation independent. However, we need to represent the variation points in the various lifecycle artifacts (domain model, code, etc.), which employ a specific representation technique. Therefore we need to map the different types of variabilities to the target notation.

The specific notation for the variation point may be graphical, text-based, or on any other basis. The different selector types should be mapped in a homogenous manner to this specification language. For each selection type a unique mapping must be found. This mapping has to take a form so that confusion with other legal expressions in the target specification language cannot occur.

Only this mapping from the selector types to the target specification mechanism must be adapted for the different formalisms. The remaining concepts can be transferred in the same manner throughout the representation mechanisms.

In order to derive a context-specific representation of variability mechanisms, we basically need to perform the following steps:

1. Determine required level of granularity,
2. Identify approach for representing variability selectors,
3. Determine supported selector types,
4. Map selector types to target notation,
5. Analyze for special representation mechanisms.

We will briefly describe these steps in more detail below.

6.1. Determine level of granularity

Prior to determining a specific approach to variability representation, we need to identify the kind of structures in the target notation, which we need to make variable. This particularly implies to identify the granularity of constructs that can be variable. Depending on the underlying notation, this can take on several forms. For example, for business process representations, this may include individual process steps, complete action sequences, or even business processes. These different levels may require specific ways for representing them. Sometimes, we will find at this point that we will actually need to support all these different levels, but even in this case this step is nevertheless important as it leads us to performing a systematic inventory of the various levels we must support. It also provides the basis for understanding the different views we must support. For example, in the context of the example on business process notations we will describe later, it was important to be able to represent variation in the flow of the business process themselves.

At the same time the underlying approach allowed us to represent organizational structures. However, the representation of variability was not relevant in this case.

6.2. Representing variability selectors

In order to represent variability based on an existing representation formalism, we must extend it in a homogenous way, so that the variability representation is well integrated. This requires a basic approach that is well compatible with the underlying notation. In particular, for formally defined representation formalisms this requires an extension of their underlying meta-model.

The most simple extension is possible in the context of text-based representations, as their underlying meta-model allows any kind of description. In this context we will use special delimiters of which we can be reasonable sure that they will not occur in the remainder of the text. In our example we will use the delimiters "<<" and ">>" to identify variability selectors. All the text that is enclosed in these "brackets" is then part of a selector. The specific type of the selector is then given by a keyword. This approach of course relies on the assumption that the delimiters will never occur in the remaining text. As obvious as this case may seem, there are certain traps involved. For example may a text-based representation mechanism include tables and figures: is it necessary to represent variability in these structures as well? If so, additional selectors must be defined. We will describe this later in our discussion of text-based approaches.

This step becomes much more difficult once we are discussing a notation with a more stringent underlying meta-model. In this case we need to find a way to extend a model in a smooth way. There are two main approaches to perform this extension:

- *Use an existing selector construct for the representation in a specific way*. We will see this later in our example for mapping variability onto code.
- *Introduce an extension to the meta-model*. This can usually be done by using an existing selector type in the specification and extending it. This establishes an analogy between existing selectors and variability selectors. It also simplifies both the definition of the variability representation as well as the later usage by the end users. We used this approach repeatedly in the context of the business process specification we will describe later.

Which of the two approaches is optimal depends on the specific notation and on existing tool support. Sometimes, tools allow for an extension of the meta-model. In this case, this approach is basically always preferable, as it can be more readily recognized on the representation level.

6.3. Select supported selector type

In Section 4.3, we described the major existing selector types. In the context of a specific modeling situation, we will usually only need a subset of these types.

In most cases, we will aim at supporting all available selector types. However, in some cases, some types may not be appropriate:

- **Optionality:** In case the underlying notation requires that all steps are required in some way (e.g., in business processes there may not occur any steps which are not

connected in any way to the main process flow), it is inappropriate to use an optionality as the removal of a step may decompose the overall flow in several unconnected sub-flows.[2]

- **Alternative:** For some system types possibilities are hardly never exclusive, but can be combined.
- **Set alternative:** This can be even rarer than the previous case. It is strongly context dependent whether this selector type is particularly useful.
- **Set selection:** For certain notations (e.g., programming languages), the set alternative must be instantiated by additionally providing some gluing mechanism for runtime. In this case this selector must either be forbidden or treated in a special way.
- **Value reference:** It strongly depends on the level of abstraction of the notation how important this mechanism is.

Based on the criteria described above, we select the selector types, that shall be supported by our variability representation mechanism.

6.4. Map selector types to target notation

The next step is to identify a notation for each of the identified selectors. This is already mainly determined by the decisions that have been made in the previous section. For example, in the case of a text-based notation, we must determine a specific keyword for the different selectors and we must define the specific notation for selecting individual alternatives.

In other notations this might be more complex (e.g., the example for graphically-based notation below), as different selector types might be more appropriately be mapped to different elements of the meta-model of the underlying notation. A typical example is, if the underlying notation provides something like an *if* and a *case* notation it is more intuitive to represent the *alternative* and *set alternative* selectors analogous to the *if* and *case* notational elements. We see this below in the example of the graphical notation where we map some selectors to new elements of the ARIS notation, while mapping others to variations of existing notational elements.

6.5. Analyze for special representation mechanisms

Finally, there might be some special notational elements that do particularly apply in the context of the target notation. These can be examples like the *optional variant decision*, which is described in our extension of the ARIS-notation discussed below. This describes basically an optional path. This path must be one of a larger set, which follow directly a runtime decision in the ARIS-notation. In addition, this selector describes the condition under which the path can be entered in case the path is present. A similar mixture of runtime and variability selector can be seen for the general variant decision.

[2] Of course, it would be possible to introduce a "repair"-semantic of optionality, e.g., the preceding and next step are simply joined. We decided against this, in order to avoid the complexities introduced by multiple links.

7. Experience

The approach to variability management in product line modeling described above has already been applied in several cases in industry. In order to provide a better understanding of the application and usefulness of this approach, we will describe some of these case studies.

In the first two case studies this was the first approach to variability management that was introduced on the specification level. In both cases the group of people who had to apply the technology was about 10–20. While there were some problems in the up-take of this technology, we found after some cooperation time that the approach was accepted. The third case study was only developed for illustration purposes.

7.1. Experiences with a text-based representation

Our variability management approach has been applied in practice with text-based requirements in an embedded systems company.

Determine required level of granularity

A textual representation was chosen because the stakeholders in the domain were very familiar with textual representations and not with other forms of requirements documents. They had also invested considerable effort into the improvement of their approach to textual requirements documentation. Therefore it was decided to base our extension of the variability management approach on the existing textual representation of requirements. The textual representation was accompanied by another representation that had to be extended: tables. Consequently, the extension of the existing representation had to take two representations into account, unstructured text and tables. We did explicitly exclude graphical representations in this context.

Identify approach for representing variability selectors

In order to be able to model and manage variability, the existing mechanisms for writing textual requirements had to be extended into a product line modeling approach. According to our approach, only the mapping of the selector types onto the target representation formalism had to be adapted. We decided to use a decision model as described in Section 4.1 that was realized using an Excel-table. A sanitized version of such a table excerpt is shown in Table 1. We used the decision evaluation primitives shown in Section 4.3. For the mapping of the selector types onto the textual specification we decided to use textual constructs framed with “<<” “>>”, as these are text fragments which so far never occurred in this domain.

To represent variability in tables we decided to integrate an extra column into each table and describe the relevancy of the row in the corresponding cell. It would also have been a possibility to use the same approach as for text, so to use constructs framed with “<<” “>>” every time a variation point occurs in a table.

Select supported selector types

We decided not to support the set alternative, as it is a special case of the set selection. Moreover, most instances we found during our work in this domain so far were instances

Table 1
Example of a decision model

Name	Relevance	Description	Range	Selection	Constraints	Binding Times
Memory	System_Mem = True	Does the system have memory?	TRUE, FALSE	1		Compile Time
Memory_Size		The amount of memory the system has (KB)	0..100.000	1	Memory=TRUE => Memory_Size > 0	Installation, System Initialisation
Time_Measure-ment		How is time measurement done?	Hardware, Software	1		Compile Time

of the set selection anyway. So, in this case we supported the selector elements optionality, alternative, set selection and value reference.

Map selector types to target notation

As we decided to use textual constructs framed with "<<" ">>" we introduced a keyword for each of the selected selector types. We described optional variability in the following way:

```
<<opt expr1 / text >>.
```

Here *expr1* is a logical expression. If it evaluates to true for a specific product (i.e., for the decision variable assignments for a specific product), then *text* is included in the instantiated product description. Similarly, for set alternative variability, we use the term:

```
<<alt expr1 / value-1 / text1
          / value-2 / text2
          .....>>.
```

Again, expr1 is a value expression, while value-1..-n are values that are in the possible range of expr1.

For set selection variability we used the same scheme. However, we found that it is usually sufficient to use a decision variable as a basis. In order to express this case we introduced the keyword mult:

```
<<mult expr1 / value-1 / text1
            / value-2 / text2
            .....>>.
```

Finally, for value references the term *<< value decision-variable >>* was used.

As in tables, the text part of the expressions could be found in the table itself, we left the *text1 and text2* part out and so had a similar representation of variability in tables.

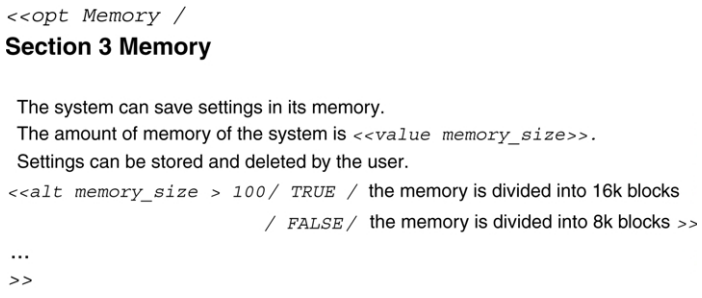

Fig. 3. An example using the textual notation.

Analyze for special representation mechanisms

To integrate specific information in the text we used the tag

<< *specific* text1 >>

Here, information that was specific for a product or for parts of the product or information and requirements that were not systematizable yet could be integrated.

By using this approach, we described the product line model. Fig. 3 shows a sanitized excerpt of such a product line model document which includes optional, alternative, and value variability. As expected, we could transfer our concepts in a straight-forward manner to this domain. During modeling we identified about 50 decision variables and about 100 variation points which had to be introduced into the documentation. We modeled the main subdomains of the system with these techniques (more than half of the system), the additional subdomains will follow in the future. We expect that once the product line model is complete, it will contain more than 100 decision variables and several hundred variation points. The resulting domain models went through inspection by the company and were well accepted by the development team. In particular the notation was considered to be well readable and the resulting models to be well understandable.

7.2. Experiences with a graphical representation

We applied our approach to variability management in the context of product line modeling in an environment, where a graphical notation was required. This notation was based on the ARIS notation [38], a business process notation method for optimizing business processes and implementing application systems, which was in turn the basis for requirements definition for systems of the customer [40]. The ARIS notation should be introduced in parallel with the variability notation in the company.

For determining the level of granularity, we had to consider the ARIS notation, which provides the basis for this application of the approach that focusses on business processes. For the second step, identification of an approach for representing variability selectors, additional elements had to be defined in order to represent also system internal information and control flows. The basic notational elements are shown in Fig. 4. Our approach focussed on augmenting this notation with additional variability elements that could be used both in business processes as well as control flow modeling.

Fig. 4. Basic notation for business processes (eEPK).

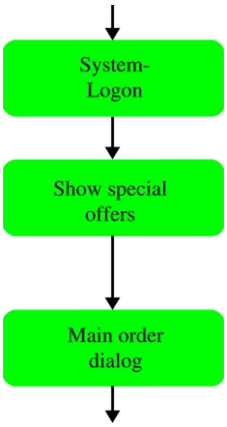

Fig. 5. An example business process.

In order to describe selections the ARIS modeling notation uses two notational elements: the connector together with the event. The *connector* defines the form of selection, the *event* defines the different cases that can occur and under what circumstances each of these paths is taken. Fig. 5 shows such an example business process with a selection.

As described in Section 4, the decision model, the interaction approach, and the relations can be taken verbatim. We only made some minor pragmatic modifications:

- The decision model had the same entries as defined above, with the exception of the binding times. There was no need to capture the binding time as this was always implicitly the modeling phase. Further there was one additional entry: the actual values for the various systems could also be defined as part of the decision description, so the instantiation of the model to products was part of the decision model here. This had pragmatic reasons, as in this case the number of systems was small. The decision model was then simply written as a web page, as much documentation in this environment was kept in an intranet-based manner.
- The decision evaluation primitives were used as described in Section 4.2.
- Regarding the different forms of variability, we decided to not support the value-reference, as we did not find a case where we would need this approach. Also the alternative is always described as a single selection.

When mapping the various variant discriminators it is key to keep in mind that we are using here a notation that imposes certain restrictions, for example, by removing some variation the overall flow may not fall apart. Thus, we can only remove certain (alternative)

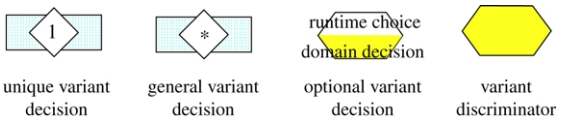

Fig. 6. Symbols added for variants.

paths from the control flow or complete business processes. This has been ensured by the specific selection and definition of the variability notation. Based on the above decisions we defined the mapping of the basic selector types onto the representation mechanisms. In order to enable the users of the approach to clearly differentiate between the basic notation and any variability information, we defined completely different notational elements, which, however, fit into the overall approach. Fig. 6 shows the different notational elements we introduced. We also adopted the differentiation between decision symbol (connector) and selectors for a specific flow (event), which is typical of ARIS.

Based on the restrictions given by the control flow, we mapped the selector types optionality, multiple selection and single selection we selected for representation, in the following manner:

Optionality: This implies that a certain path may either be part of a system variant (an instantiation), or not. Thus, we need to attach two forms of information to it: the situation in which this path is part of the final model and if it is part of the final model, the (runtime) situation in which it is actually taken. The second part obviously corresponds to the event mechanism in the ARIS business modeling approach, while the former is the optionality-specific addition. We thus added the optional variant decision to the modeling language. As shown in Fig. 6 it consists of a runtime decision and a domain decision part. The domain decision part in turn uses the decision evaluation primitives as described in Section 4.2 in order to describe whether the branch started with this decision should be part of the instantiated model. The runtime decision part in turn is annotated using the ARIS-notation in order to describe what will happen in case this branch is selected.

Single selection: The single selection is mapped to the unique variant decision (cf. Fig. 6). This works similar to a runtime decision in ARIS, with replacing connectors by the unique variant decision and the events to the variant discriminator (cf. Fig. 7). In this case we restricted the expression for selecting among the various paths to a decision variable, with the variant discriminators showing the different values. Note that upon resolution of the variability none of the notational symbols for variability will remain in the instantiated flow.

Multiple selection: The multiple selection has been mapped in very much the same way as the single selection. The main reason for having both of them was clarity of the instantiation semantics. In a work flow (or control flow) representation like ARIS, a runtime decision must remain upon resolution of the specification variability in the case of a multiple selection.[3] This is different from the single

[3] Note that in the context of this case study [1], we differentiated only between specification time and runtime.

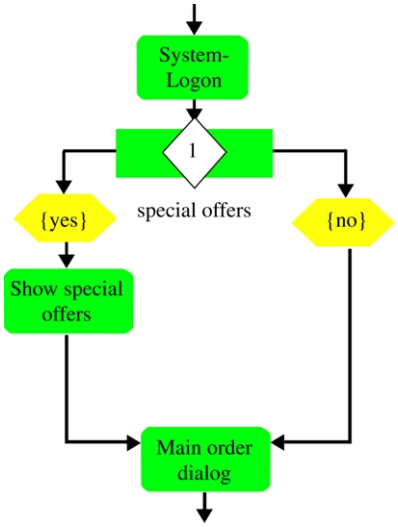

Fig. 7. Example for the description of variants.

selection where all variability symbols are removed upon instantiation. Here, they are transformed into runtime variability (if more than one option is chosen) and so stay as decisions in the product and will not be bound at a binding time before.

This approach to modeling was used for modeling several systems in the domain of merchandising information systems and a product line of about ten e-commerce shops. We found this approach to be easily applicable to these systems. Especially in the e-commerce context it was also well accepted by the development personnel.

7.3. UML-based representation

Different extensions to the UML or to parts of the UML have already been made by others [7,17,20,45]; this includes extensions of use cases or extensions of architecture/design related parts of UML like class diagrams. Also more general approaches to extending UML have been described [2,14].

We have developed a notation that complies to our variability modeling approach by extending use case diagrams (see Fig. 8) and textual use cases following the notation of Cockburn [11]. This extension was already described in [25,26]. We used this notation in two case studies. In one case study we applied product line concepts on an example product line of mobile phones throughout the lifecycle in the context of the ViSEK Project (a Virtual Software Engineering Competence Center with emphasis on empirical studies) [47]. In the second case study we modeled the requirements on a variable cruise control (for an overview see Fig. 8) with different approaches like Use Cases, Features [27] and Design Spaces [6]. When developing this representation-specific approach we first had to determine the level of granularity. As we wanted to extend mechanisms used for requirements specification, we decided to extend use case diagrams and textual use cases

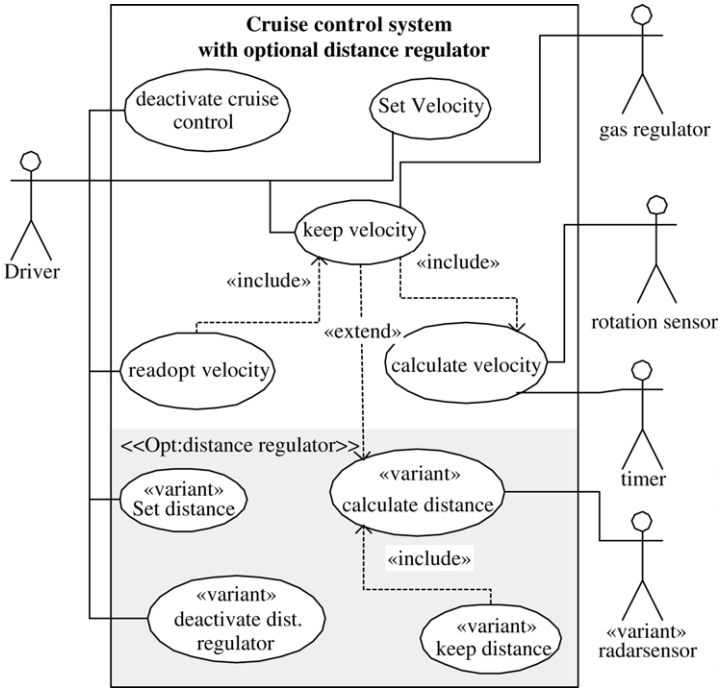

Fig. 8. Use case diagram with variability.

in order to have an overview and a more detailed description mechanism for requirements in use cases.

An important requirement for identifying the approach for representing variability selectors was that the extension is supported by common UML-modeling tools like Visio or Rational Rose and that the textual extension can easily be integrated into a word processor. We decided to use stereotypes for the extensions of Use Case diagrams and XML-like tags for extending the textual use cases. Also optional and alternative selector types in the diagrams and optional, alternative and set selection and value reference in the textual use cases are supported. The mapping of the selector types to the target notation UML was given by using the different stereotypes <<opt>> for optionality, <<alt>> for alternative and <<variant>> for variable parts. Variability was distinguished in local variability (variability within a use case) and global variability (whole variable parts, like the grey <<opt distance regulator>> within an optional or alternative part of the use case diagram (see Fig. 8). For the textual representation we used tags like

<variant OPT> text1 *</variant>*

As this notation is similar to the notation described in Section 7.1, we will not go into further detail here.

7.4. Implementation representation

Similarly to textual or graphical mapping a mapping to an implementation can be done. As an example, we use the mapping to a compile-time binding, based on the C language. The obvious approach for this is to use the C preprocessor. The preprocessor provides macro capabilities that can be used to select the code that is compiled. This language is very restricted. It allows us to use *#define* to define a macro (which may contain parameters). It also allows us to test for complex expressions using *#if* and whether a variable is defined (*#ifdef*). In addition an *#include* directive allows us to include a large part of C code at the corresponding position.

Following our process, we first have to determine the required level of granularity. Possible levels of granularity could be file level, function level, statement, or even sub-statement level. According to our experience, usually, we will need to be able to support all of these. To represent variability and variability selectors, different approaches exist. In a C environment, it is usually necessary to support all levels, which basically requires to use the preprocessor as a basis to implement fine-grained variability. In particular, we can, as we show here, represent individual decision variables as precompiler variables (e.g., Memory_Size and Time_Measurement in Fig. 9).

This technique enables us to map the decision model itself to preprocessor directives also. The relevancy-section of a decision variable can in this case be mapped by defining the preprocessor variable only if the corresponding decision variable is relevant (not shown in the example). The constraints will usually only be mapped, if they lead to a unique value for a decision variable, as in this case it can be automatically be defined.

The description of dependences on the decision variable values for a specific product is then described using the *#if* directive. So we can map in a straightforward manner optionality and alternative. In order to map a set alternative we can use the *#elif* construct as shown in Fig. 9. The set selection can not be directly translated. Rather, we actually need to break it down into the different specific cases. Finally, the value reference can be used in a straightforward manner, as this is automatically resolved by the preprocessor. In addition, it is common to use the *#ifdef* directive as a shorthand notation in the context of boolean decisions.

In Fig. 9, we illustrate the translation of an optionality (Time_Measurement) and of a set alternative (Memory_Size). Both examples are based on the decision model given in Table 1. As the example illustrates, the translation into the C preprocessor language can be cumbersome and the expressiveness of the C preprocessor may impose some restrictions. This can be improved by using more powerful preprocessors. To some extent also constraints from the decision model can be implemented using the preprocessor. However, in our example, we simply assume that permissible values are initially provided to the preprocessor.

8. Conclusion

In this paper we have described an approach to variability modeling in a product line context. The development of this approach was driven from the need for an approach that

```
#if (Time_Measurement = Software)
start_software_clock();
#else
start_hardware_clock();
#endif

#if Memory_Size == 0
get_no_memory();
#elif Memory_Size == 10
get_small_memory();
#elif Memory_Size = 100
get_medium_memory();
#else /* Memory_Size = 1000 */
get_large_memory();
#endif
```

Fig. 9. An example using the C preprocessor.

can be easily applied in a wide range of practical contexts and in combination with many different specification techniques.

The key contribution of this approach is that it provides a general framework that allows the introduction of variability management into a wide range of existing notations. Thus, it allows us to separate the dimensions of the adequate representation of the content from the description of the variability. As an additional benefit, the genericity of the underlying approach allows us to use it throughout the different lifecycle stages in an integrated manner. The usage of the decision model supports the comprehensive treatment of variability across the lifecycle stages. In particular, it enables traceability of the variability concerns through the different lifecycle stages. The planned evolution in the space of the products can be coped with by the decision model, for evolution over time we use configuration management techniques [30]. This integration of our variability management approach and configuration management approaches is still an open point and should be investigated in the future.

The approach has already been applied in some industrial applications, supporting its real-world applicability although there has been no tool support other than standard tools like Microsoft Visio, Excel and Word. We expect to perform more applications in industrial settings in the future. They will further demonstrate the applicability of the approach throughout all lifecycle stages.

Acknowledgements

This work was supported in part by Eureka Σ! 2023 Programme, ITEA project ip00004, Café.

References

[1] M. Anastasopoulos, C. Gacek, Implementing product line variabilities, in: Proceedings of the 2001 ACM SIGSOFT Symposium on Software Reusability, SSR 2001, Canada, May, 2001, ACM Press, New York, 2001.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel, Component-based Product Line Engineering with UML, Component Software Series, Addison-Wesley, 2001.
[3] J. Bayer, D. Muthig, T. Widen, Customizable domain analysis, in: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering, GCSE'99, Erfurt, Germany, September, 1999.
[4] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud, PuLSE: a methodology to develop software product lines, in: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability, SSR'99, Los Angeles, CA, USA, May, 1999.
[5] M. Becker, Towards a general model of variability in product families, in: Proceedings of the First workshop on Software Variability Management, Groningen, February, 2003.
[6] M. Becker, L. Geyer, A. Gilbert, K. Becker, Comprehensive Variability Modelling to Facilitate Efficient Variability Treatment Fourth International Workshop on Product Family Engineering, PFE-4, Bilbao, Spain, October, 2001.
[7] A. Bertolino, A. Fantechi, S. Gnesi, G. Lami, A. Maccari, Use case description of requirements for product lines, in: Proceedings of the International Workshop on Requirements Engineering for Product Lines, REPL'02, September, 2002.
[8] J. Bosch, Design and Use of Software Architectures, Addison-Wesley, 2000.
[9] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl, Variability issues in software product lines, in: Proceedings of the Fourth International Workshop on Product Family Engineering, PFE-4, Bilbao, Spain, October, 2001.
[10] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, 2001.
[11] A. Cockburn, Writing Effective Use Cases, Addison Wesley, 2001.
[12] CONIPF Project Page http://dbs.cordis.lu/fep-cgi/srchidadb?ACTION=D&CALLER=PROJ_IST&QF_EP_RPG=IST-2001-34438.
[13] M. Coriat, F. Waeber, Product line process framework: the wheels process, in: Proceedings of the International Workshop on Software Product Lines: Economics, Architectures, and Implications, Limerick, Ireland, June, 2000.
[14] M. Coriat, J. Jourdan, F. Boisbourdin, The SPLIT method, in: P. Donohoe (Ed.), Proceedings of the First Software Product Line Conference, Kluwer Academic Publishers, 2000, pp. 147–166.
[15] J.-M. DeBaud, K. Schmid, A practical comparison of major domain analysis approaches—towards a customizable domain analysis framework, in: Proceedings of the Tenth Conference on Software Engineering and Knowledge Engineering, SEKE'98, June, 1998.
[16] M. Eisenbarth, Domain-specific customizations for product line modelling, Diploma Thesis, Fraunhofer IESE and University of Kaiserslautern, 2003.
[17] O. Flege, System family architecture description using the UML, Technical Report IESE Report No. 092.00/E, Fraunhofer IESE, 2000.
[18] C. Fritsch, A. Lehn, T. Strohm, Evaluating variability implementation mechanisms, in: Proceedings of the Second International Workshop on Product Line Engineering—The Early Steps: Planning, Modeling, and Managing, PLEES'02, November, 2002.
[19] A. Günter, C. Kühn, Knowledge-based configuration—survey and future directions, in: Proceedings of XPS 99—Knowledge Based Systems, Springer LNAI, vol. 1570, 1999.
[20] G. Halmans, K. Pohl, Communicating the variability of a software-product family to customers, Journal of Software and Systems Modelling SoSyM 2 (2003).
[21] L. Hotz, T. Krebs, Configuration—state of the art and new challenges, in: Proceedings of PuK2003, vol. 17, Workshop "Planen, Scheduling und Konfigurieren, Entwerfen" Hamburg, 15–16 September, 2003.
[22] L. Hotz, A. Günther, T. Krebs, A Knowledge-based product derivation process and some ideas how to integrate product development, in: Proceedings of the First Workshop on Software Variability Management, Groningen, February, 2003.
[23] I. Jacobson, M. Griss, P. Jonsson, Software reuse, in: Architecture, Process and Organization for Business Success, Addison-Wesley, 1997.
[24] U. John, U. Geske, Constraint-based configuration of large systems, in: Web Knowledge Management and Decision Support, LNCS, vol. 2543, Springer Verlag, 2003.

[25] I. John, D. Muthig, Tailoring use cases for product line modeling, in: Proceedings of the International Workshop on Requirements Engineering for Product Lines, REPL'02, September, 2002.
[26] I. John, D. Muthig, Modeling variability with use cases, Technical Report IESE Report No. 063.02/E, Fraunhofer IESE, 2002.
[27] K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
[28] M. Kasunic, Synthesis: A reuse-based software development methodology, Process Guide, Version 1.0. Technical Report, Software Productivity Consortium Services Corporation, October, 1992.
[29] C. Krueger, Variation management for software product lines, in: G. Chastek (Ed.), Proceedings of the Second Software Product Line Conference, San Diego, CA, August, LNCS, vol. 2379, Springer, 2002.
[30] R. Laqua, Concepts for a product line knowledge base & variability, in: Proceedings of NetObjectDays 2002, Erfurt, October, 2002.
[31] M. Mannion, B. Keepence, H. Kaindl, J. Wheadon, Reusing single system requirements for application family requirements, in: Proceedings of the 21st International Conference on Software Engineering, ICSE'99, May, 1999.
[32] D. McComas, S. Leake, M. Stark, M. Morisio, G. Travassos, M. White, Addressing variability in a guidance, navigation, and control flight software product line, in: Architecture Workshop of the First Software Product Line Conference, August, 2002.
[33] S. Mittal, B. Falkenhainer, Dynamic constraint satisfaction problems, in: Proceedings of AAAI-90, AAAI Press, Boston, USA, 1990.
[34] M. Morisio, G. Travassos, M. Stark, Extending UML to Support Domain Analysis, ASE'00, Grenoble, France, 11–15 September, 2000.
[35] D. Muthig, A light-weight approach facilitating an evolutionary transition towards software product lines, Ph.D. Theses in Experimental Software Engineering, Fraunhofer IRB Verlag, 2002.
[36] D. Muthig, T. Patzke, Generic implementation of product line components, in: Proceedings of the Net.ObjectDays, NODE'02, Erfurt, Germany, October, 2002.
[37] Object Management Group. OMG Unified Modeling Language Specification, Version 1.4, September 2001.
[38] A. Scheer, ARIS—Business Process Frameworks, 3rd edition, Springer, 1999.
[39] K. Schmid, I. John, Generic variability management and its application to product line modelling, in: Proceedings of the Variability Management Workshop in Groningen, 2003.
[40] K. Schmid, U. Becker-Kornstaedt, P. Knauber, F. Bernauer, Introducing a software modeling concept in a medium-sized company, in: Proceedings of the 22nd International Conference on Software Engineering, ICSE 2000, Limerick, Ireland, 2000.
[41] Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC, Reuse-Driven Software Processes Guidebook, Version 02.00.03, November, 1993.
[42] Software Technology for Adaptable, Reliable Systems (STARS), Organization Domain Modeling (ODM) Guidebook, Version 2.0, June, 1996.
[43] M. Svahnberg, J. van Gurp, J. Bosch, A Taxonomy of Variability Realization Techniques, Technical Paper, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.
[44] S. Thiel, A. Hein, Systematic integration of variability into product line architecture design, in: Proceedings of the Second Software Product Line Conference, SPLC2, August, 2002, San Diego, CA, LNCS, vol. 2379, Springer, 2002.
[45] T. van der Maßen, H. Lichter, Modeling variability by UML use case diagrams, in: Proceedings of the International Workshop on Requirements Engineering for Product Lines, REPL'02, September, 2002.
[46] J. van Gurp, J. Bosch, M. Svahnberg, On the notion of variability in software product lines, in: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, WICSA'01, 2001.
[47] ViSEK Page of the GoPhone Case Study, http://www.softwarekompetenz.de/?11174.
[48] S. Voget, Product line variability in the automotive domain, Proceedings of the International Colloqium of SFB501, Fachbereich Informatik, Universität Kaiserslautern, March, 2003, http://www.sfb501.uni-kl.de/news/IntColloquium_SFB501_March_2003.pdf.
[49] D.M. Weiss, C.T.R. Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999.